# Python Printing, Scripts, and Functions

## 1 Purpose

- Use the `.format()` string method to generate specifically formatted output
- Use the `input()` function to generate interactive input from the user in scripts and user-defined functions
- Create, edit, and execute simple scripts using *Python*
- Assign values to variable names within scripts
- Request user input to assign values to variables in scripts using the `input()` function
- Create and execute user-defined functions that do and do not accept arguments
- Create and execute void and fruitful functions
- Use `print()` to display output from scripts and user-defined functions

## 2 Some Creative Commons Reference Sources for This Material

- *Think Python 2nd Edition*, Allen Downey, chapters 3 and 6
- *The Coder's Apprentice*, Pieter Spronck, chapters 5 and 8
- *A Practical Introduction to Python Programming*, Brian Heinold, chapters 1, 10, 13, and 23
- *Algorithmic Problem Solving with Python*, John Schneider, Shira Broschat, and Jess Dahmen, chapters 3 and 4

## 3 Reviewing the `print()` Function

- `print()` can display numeric values and text strings
- Multiple items can be printed by separating items with commas
- Force a line return in any string by adding the newline escape sequence `\n`
- The escape sequence `\t` adds a tab
- Multiplying a string by an integer in `print()` will print the string that number of times

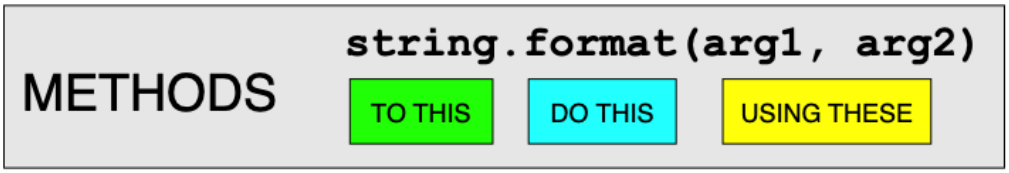
Review of `print()`

```
>>> print("Python is awesome")
Python is awesome
>>> print('Lumberjacks', "Parrots", 42)
Lumberjacks Parrots 42
>>> print(4*5)
20
>>> print(2*'Hello')
HelloHello
>>> print('Hello,\nWorld!')
```

```
Hello,
World!
```

## 4 Functions Versus Methods

- *Python* uses both **functions** and **methods** to work with/on objects
- Functions usually work with arguments to return a value or do something with the arguments
  - Like the `abs(x)` function
  - Or like `print()`
- Methods are similar to functions
  - Methods work on an object using arguments to either return a value or change the object
  - Can accept arguments but often don't
  - Syntax is different than functions
  - The string `.format()` method will be investigated at this time
- See images below


## 5 Formatting Printed Output

### 5.1 The `.format()` String Method

- Including the `.format()` string method within a `print()` function
- Control exactly how numeric (and non-numeric) values are displayed
- The general layout is `'The sum of {} and {} is {}'.format(item_1, item_2, item_3)`
- The expression starts with a string that has curly braces `{}` as placeholders
- Think of the curly braces as blanks in a fill-in-the-blanks statement
- The `.format()` method directly follows the closing quote for the string
- The arguments in the parentheses are the values (in order) that match the placeholders in the string
- Arguments may be values and/or expressions
- Placeholders can include formatting descriptors to generate very specific formatting
- Formatting descriptors within braces must be preceded by a colon, i.e. `{:.2f}`
- https://pyformat.info has many examples of the `.format()` method; both simple and complex

**Examples of using `.format()`**

The result of `22/7` is an estimate for $\pi$ that has historically been used by many tool makers and machinists. The following examples use `print()` and the `.format()` string method.

Using `print()` with two arguments and a comma but no formatting

```
>>> # Standard print() without .format()
>>> print('pi is close to', 22 / 7)
pi is close to 3.142857142857143
```

`{}` – No specific formatting assigned

```
>>> print('pi is close to {}'.format(22/7))
pi is close to 3.142857142857143
```

`{:f}` – Standard floating point notation with default decimal places

```
>>> print('pi is close to {:f}'.format(22/7))
pi is close to 3.142857
```

`{:.5f}` – Standard floating point notation with 5-decimal places

```
>>> print('pi is close to {:.5f}'.format(22/7))
pi is close to 3.14286
```

## 5.2 Formatted String Literals

- New to *Python* starting with version 3.6
- Also called "*f-strings*"
- They work like the `.format()` method but in a more direct way
- Use `print(f'pi is close to {22/7}')` instead of `print('pi is close to {}'.format(22/7))`
- Allow expressions or variables to be placed directly within the curly braces
- Use a colon is added after the expression or variable to add a formatting descriptor, i.e. `print(f'pi is close to {22/7:.8f}')`

**Formatted string literal (*f-string*) examples**

`:f` – Standard floating point notation with default decimal places

```
>>> print(f'pi is close to {22/7:f}')
pi is close to 3.142857
```

`:.16f` – Standard floating point notation with 16-decimal places

```
>>> print(f'pi is close to {22/7:.16f}')
pi is close to 3.1428571428571428
```

`:e` – Exponential notation (`:E` for uppercase)

```
>>> print(f'pi is close to {22/7:e}')
pi is close to 3.142857e+00
```

`:.12E` – Exponential notation with 12-decimal places

```
>>> print(f'pi is close to {22/7:.12E}')
pi is close to 3.142857142857E+00
```

`:g` – Standard or exponential notation whichever is more efficient (`G` for uppercase)

```
>>> print(f'pi is close to {22/7:g}')
pi is close to 3.14286
```

`:.12g` – 12-decimal places and automatically use standard or exponential notation

```
>>> print(f'pi is close to {22/7:.12g}')
pi is close to 3.14285714286
```

`:8.3f` – Total width of 8 characters with 3 to the right of the decimal

```
>>> print(f'pi is close to {22/7:8.3f}')
pi is close to    3.143
```

- `8` => 8 total characters set aside
- `.3` => 3 digits to the right of the decimal point
- `f` => the value will be formatted as a float
- Decimal point counts as a character
- Leading blanks are added as needed

Formatting guide

```
| | | |3|.|1|4|3|  <= formatted value
| | | | | | | | |
|8|7|6|5|4|3|2|1|  <= characters set aside for the value
```

`:+08.3f` – Same as the previous but with leading zeros and the ± sign

```
>>> print(f'pi is close to {22/7:+08.3f}')
pi is close to +003.143
```

`:.0f` – Force a floating point to display no values right of the decimal (like an integer)

```
>>> print(f'pi is close to {22/7}')
pi is close to 3.142857142857143
```

Using a variable in an f-string

```
>>> almost_pi = 22/7
>>> print(f'pi is close to {almost_pi:.8f}')
pi is close to 3.14285714
```

`:d` – Standard integer (object must be of `int` type)

```
>>> print(f'Integer value: {42:d}')
Integer value: 42
```

`:4d` – Integer with 4 total characters

```
>>> print(f'Integer value: {42:4d}')
Integer value:   42
```

`:04d` – Integer with 4 total characters and leading zeros

```
>>> print(f'Integer value: {42:04d}')
Integer value: 0042
```

`:+04d` – Integer with 4 total characters and leading zeros and the ± sign

```
>>> print(f'Integer value: {42:+04d}')
Integer value: +042
```

`:,d` – Integer with comma separators

```
>>> print(f'Integer value: {987654321:,d}')
Integer value: 987,654,321
```

`:s` String (although setting the formatting is not really necessary in this case)

```
>>> first_name = "Slim"
>>> last_name = "Shady"
>>> print(f'I am the real {first_name:s} {last_name:s}')
I am the real Slim Shady
```

`:^10s` – String centering in 10 characters (use < and > for left/right justified)

```
>>> first_name = "Slim"
>>> last_name = "Shady"
>>> print(f'I am the real {first_name:^10s} {last_name:^10s}')
I am the real    Slim      Shady
```

## 6 The `input()` Function

- Used to request information from a user at a command line in a script
- Accepts one optional argument
  - A statement or question to the user so they know what to enter
  - Must be a string or formatted string
- `input()` always returns a string
  - Must specifically convert the returned value to an integer or float if that is desired
  - Last two examples have the `input()` function inside of `float()` and `int()`
- Good practice to end prompt strings with a delimiter of some sort and a space
- Common delimiters
  - Question mark (`?`)
  - Colon (`:`)
  - Right arrow (`>`)
- The delimiter and space makes it easier to read, so please *do it*

Sample `input()` statements

```
user_name = input('What is your name? ')
city = input("Enter your city of residence: ")
applied_load = float(input('Enter the applied load (lbf): '))
age = int(input('Enter your current age > '))
```

# 7 Scripts and User-Defined Functions

Quite often it is more efficient to use a script or user-defined function than to enter commands at a REPL prompt or in a *Jupyter* code cell. A script can be used on any computer with *Python* installed. Functions are usually written as part of a script of module. When a function is written within a script it is called/used in the script's code. A script is referred to as a module if it just contains functions. Calling functions that are in modules requires importing the module file then calling the function (see the examples below).

Running scripts and calling functions from a *Python* prompt

```
>>> # Running a script from a Python prompt
>>> import hello
Hello, World!

>>> # Importing module then calling functions from it
>>> import my_module
>>> my_module.my_function()
Nice function

>>> # Importing a specific function from a module
>>> from my_module import cuberoot
>>> cuberoot(27)
3.0
```

## 7.1 Script Details

- Scripts are sometimes referred to as programs
- Simple scripts are no more than lists of commands that are executed sequentially from top to bottom
- Receive necessary values using several methods
  - Assign to variables directly ("hard-coded")
  - Ask the user to enter values at a prompt using the `input()` function
  - Pass values to the script when it is executed
  - Load values from a file
- Assigned variables in scripts are available for use after being assigned
- Results from scripts can be made available via...
  - The `print()` function
  - Writing output values to a text file
- Use good programming practices
  - Write a docstring-style comment at the top of the script
  - Place all `import` statements at the beginning of scripts
  - Place all user-defined functions immediately after imports
  - Clean up your code with a style formatter like *Black* or *autoPEP8*
  - Include comments throughout explaining "why"

## 7.2 User-Defined Function Details

- Contained within script or module files
- Must be defined before they can be called (used)
- Usually receive input by passing arguments instead of using `input()` commands
  - `abs(-100)` *passes* `-100` as an argument to the absolute value function
- Usually return results back to the calling location instead of printing
  - `abs(-100)` *returns* the value `100` but does not print the result
- Argument names passed into and used in a function are only available for use in that function
- Variables created inside a function are only available inside that function
- Can be written to return results, print results, or return and print results
- Functions that don't return values are sometimes called void functions
- Functions that do return values are sometimes called fruitful functions
- Follow good programming practices (see above)

## 7.3 Creating, Editing, and Executing a Script

- Plain text files written using text editors
  - *Jupyter* environment includes a simple text editor with syntax highlighting
  - *VSCode* is a good, free text editor for programming in *Python*
- *Python* script files end with a `.py` extension
- Execute scripts from...
  - Command line prompt
    - `python script_name.py`
    - `python3 script_name.py`
  - *Jupyter* code cell
    - `run my_script.py`
    - Script must be in the same directory as the notebook
  - *Python* prompt
    - `import my_script`
- Good practice to include comments at the top of a script file
  - Describing what the script does
  - Units that are used
  - Special conditions, etc.
  - Should include a comment with your name, website, licensing, etc.

### Example script with hard-coded values

Create a new text file named `mph2kph.py`. It is important that there are no spaces before, after, or within the script name and that the extension is included. The script as converts from mph to km/h. The value of `mph` will be "hard-coded" instead of using an `input()` for this script.

*Python* script – `mph2kph.py`

```
"""
this script converts mph to km/h
the conversion factor is 1 mph = 1.609 km/h

author: Brian Brady
        Ferris State University
"""

# assign a value to a variable named 'mph'
mph = 60

# assign the value 1.609 to a variable named 'conversion'
conversion = 1.609

# calculate and assign the speed in km/h to the variable 'kph'
kph = mph * conversion

# use a print() function to create output similar the following example:
# 60 mph is 96.53999999 km/h
print(f"{mph} mph is {kph} km/h")

# end of script
```

Running `mph2kph.py`

```
>>> import mph2kph
60 mph is 96.53999999999999 km/h
```

### Example script with `input()`

The previous script can be improved with interactive user input. Instead of assigning a fixed value to `mph`, an `input()` function can be used to ask the user to enter a speed in mph. Placing the `input()` function inside the parentheses of a `float()` function will convert the input value from a string into a float so calculations can be performed with the input value.

The `print()` expression will be changed to use formatted output that displays the input speed with no special formatting and the calculated speed with one decimal place.

*Python* script with user input – `mph2kph_input.py`

```
"""
this script converts mph to km/h
user enters mph when prompted
the conversion factor is 1 mph = 1.609 km/h

author: Brian Brady
        Ferris State University
"""

mph = float(input("Enter a speed in mph: "))
conversion = 1.609
kph = mph * conversion

print(f"{mph} mph is {kph:.1f} km/h")

# end of script
```

### Another example script

The following script named `windchill.py` uses `input()` functions to ask the user to enter the air temperature in degrees F and the wind speed in mph then calculates the wind chill temperature $T_{wc}$ in degrees F.

$$T_{wc} = 35.74 + 0.6215\,T - 35.75\,v^{0.16} + 0.4275\,T\,v^{0.16}$$

The statement **"xxx F with a yyy mph wind equals a zzz F wind chill"** will be displayed using formatted printing such that the input values have no special formatting and the calculated value is a float with zero decimal places. An introductory docstring-style comment describes what the script does, the units being used, and the author's name (similar to the previous scripts). Testing the script with the following values will yield the results shown.

- 30°F with 10 mph ==> 21°F
- 10°F with 30 mph ==> −12°F
- −10°F with 20 mph ==> −35°F

*Python* script – `windchill.py`

```
"""
Determines windchill in degrees F
Inputs are air temperature in degrees F and wind speed in mph

Author: Brian Brady, Ferris State University
"""

T = float(input("Enter air temperature in F: "))
v = float(input("Enter wind speed in mph: "))

Twc = 35.74 + 0.6215*T - 35.75*v**0.16 + 0.4275*T*v**0.16

print("{T} F with a {v} mph wind equals a {Twc:.1f} F wind chill")
```

## 8 Importance of Indentation

Indentation (spaces at the beginning of lines) is important in *Python*. The most important thing to remember about indentation is that it must be consistent. Spacing between objects in a line of code is generally not important however.

- In the previous script examples all lines were aligned on the left edge
- Indentation is used in *Python* to group commands for specific purposes
- The default indentation is no indentation at all
- Even a single extra space at the beginning of a line will generate an error (see below)
- Standard indentation is 4 spaces (when indentation is required)
- Spaces and tabs for indentation must not be mixed

Indentation is important

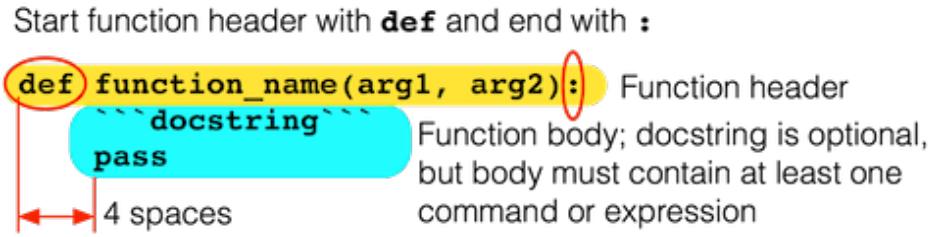
```
>>> print("This line is fine")
This line is fine

>>> # The following line has a space at the beginning
>>>  print("The space at the start of this line causes an error")
  File "<stdin>", line 1
    print("The space at the start of this line causes an error")
    ^
IndentationError: unexpected indent

>>> # Spaces between objects are ignored
>>> 25 / 5       + 2       *3
11.0
```

## 9 User-Defined Function Structure

- Indent all commands that belong to the function by 4-spaces except the header
- Indenting signals to *Python* that the lines belong to the function definition




- The first line is the function header and includes...
  - Function name
  - Any arguments to pass in parentheses
  - End with a colon
- Lines below the header are the function body
  - Must consist of at least one command
  - `pass` command may be used as a placeholder if there are no other commands
  - A `pass` command tells *Python* to exit the function and go back to where it was
- Function definitions may optionally include a docstring
  - Displays information about the function when `help()` is called with the function name
  - The docstring must be enclosed by a set of three double quotes
  - Can span multiple lines if needed, but just one set of double quotes total

**Example of a user-defined function definition and calling**

Simple function definition

```
def do_nothing():
    """
    This function does nothing useful.
    Arguments: no arguments accepted
    Prints: "Nothing done"
    Returns: does not return anything
    """

    print("Nothing done")
```

Calling and getting help on a user-defined function

```
>>> do_nothing()
Nothing done

>>> help(do_nothing)
Help on function do_nothing:

do_nothing()
    This function does nothing useful.
    Arguments: no arguments accepted
    Prints: "Nothing done"
    Returns: does not return anything
```

## 9.1 Void Functions

- Void functions do not return any values
- They may print something, but don't return anything
- The above function, `do_nothing()`, is a void function

### Example void function with one argument

The following code block contains another void function that accepts a single argument and prints the result of the argument value multiplied by 2.

Void function – `double_me(value)`

```
def double_me(value):
    """
    This function multiplies the argument named value by 2
    and prints the result
    """
    doubled_value = value * 2
    print(doubled_value)
```

Calling `double_me(value)`

```
>>> double_me(10)
20
```

### Example void function with multiple arguments

This next example is also a void function, but it accepts two arguments. Notice that the argument names used in this and the previous example are not the same. You can use any valid variable name as a function argument name. Also note that the calculation is performed directly in the `print()` function.

Void function – `multiply2(arg1, arg2)`

```
>>> def multiply2(arg1, arg2):
...     print(arg1 * arg2)
...
>>> multiply2(6, 7)
42
```

### Example of void function with formatted printing

The following example defines and calls a function named `print_mph2kph` that accepts one argument named `mph`. The function converts `mph` to km/h and assigns the result to the name `kph`. On the last line of the function the result is printed such that it reads "xxx mph equals yyy km/h", where "xxx" and "yyy" are replaced by values of "mph" and "kph".

Void function – `print_mph2kph(mph)`

```
>>> def print_mph2kph(mph):
...     kph = mph * 1.609
...     print(f"{mph} mph equals {kph} km/h")
...
>>> print_mph2kph(60)
60 mph equals 96.53999999999999 km/h

>>> print_mph2kph(42)
42 mph equals 67.578 km/h
```

## 9.2 Fruitful Functions

- Fruitful functions return a value or values back to the caller
- They usually do not create any printed output
- Must have a `return` statement
- The `return` is usually located on the last line since a function quits once a value is returned
- Return multiple values by separating them with commas after the `return` statement
- `return` is a statement, not a function, and does not use parentheses
- The example below illustrates the general structure of a fruitful function

General structure of a fruitful function

```
def my_function(arg1, arg2):
    """docstring"""
    body line 1
    body line 2
    return value1, value2
```

### Example fruitful function

The following function definition for `double_me2` is a modification of the `double_me` function from earlier. This time, instead of printing the result of the calculation, the function returns the result.

Fruitful function – `double_me2(value)`

```
def double_me2(value):
    """
    This function multiplies the argument named value by 2
    and returns the result
    """
    doubled_value = value * 2
    return doubled_value
```

Calling `double_me2(value)`

```
>>> double_me2(10)
20
>>> two_times = double_me2(21)
>>> print(two_times)
42
```

A more *Pythonic* way to define the above function would be to move the calculation into the `return` line. Unless the intermediate variable is needed for another calculation in the function, it is more efficient to just return the calculation directly.

Fruitful function – `double_me3(value)`

```
def double_me3(value):
    """
    Doubles value and returns it
    """
    return value * 2
```

### Example fruitful function with 2 arguments and 2 returned values

The following `rectangle(width, height)` function returns the area and perimeter of a rectangle with sides of `width` and `height`. The two returned values can be assigned to two variables when the function is called like so...

```
(area, perim) = rectangle(width, height)
```

The grouping `(area, perim)` is referred to as a **tuple** by *Python* and is the *Pythonic* way of assigning multiple values to multiple variable names at the same time.

Fruitful function – `rectangle(width, height)`

```
>>> def rectangle(width, height):
...     """
...     Returns the area and perimeter (in that order) for a
...     rectangle of width and height
...     """
...     area = width * height
...     perimeter = 2 * width + 2 * height
...     return area, perimeter
...
>>> rectangle(5, 8)
(40, 26)
>>> area, perim = rectangle(10, 24)
>>> print(f"Area = {area} and perimeter = {perim}")
Area = 240 and perimeter = 68
```

### Another example fruitful function

The previously created `print_mph2kph` function has been recreated below such that it returns the speed in km/h but does not print anything.

Fruitful function – `return_mph2kph(mph)`

```
>>> def return_mph2kph(mph):
...     return mph * 1.609
...
>>> return_mph2kph(55)
88.495
>>> speed = return_mph2kph(75)
>>> print(speed)
120.675
```

### Yet another example fruitful function

The following function named `windchill(tempF, vel_mph)` does the same thing as the script from earlier called `windchill.py` except instead of using `input()` functions to get the air temperature and wind speed it uses two arguments called `tempF` and `vel_mph`. The resulting wind chill temperature is rounded to zero decimal places when it is returned.

$$T_{wc} = 35.74 + 0.6215\,T - 35.75\,v^{0.16} + 0.4275\,T\,v^{0.16}$$

Testing the function with the following values results in 15°F, 4°F, and −4°F.

- 30°F with 30mph
- 20°F with 20mph
- 10°F with 10mph

Fruitful function – `windchill(temp_F, vel_mph)`

```
def windchill(temp_F, vel_mph):
    """
    Calculates and returns windchill temperature in degrees F
    Arguments: temp_F - air temperature in degrees F
               vel_mph - wind speed in mph
    """
    T = temp_F
    v = vel_mph
    T_wc = 35.74 + 0.6215*T - 35.75*v**0.16 + 0.4275*T*v**0.16
    return round(T_wc)
```

Calling `windchill(temp_F, vel_mph)`

```
>>> windchill(30, 30)
15
>>> windchill(20, 20)
4
>>> windchill(10, 10)
-4
```